Python Dictionaries and Files

Main Points

1. Remember lists and tuples?
2. What are *Python* dictionaries?
3. Getting values from dictionaries
4. Adding and replacing values
5. Dictionaries can be nested
6. Dictionaries have methods
7. Sets are similar to dictionaries
8. Working with files

1 Review of Lists and Tuples

Recall that lists and tuples can be used to hold a number of objects or values of different types, such as the list `my_list = [1, 2, 3, 4, "Python", 10.3, 56.78, math.pi, (3, 4, 5)]`.

- Access the items in lists (and tuples) using indexing and slicing
  - Use `my_list[4]` to access the string `"Python"`
  - Use `my_list[:4]` to access `[1, 2, 3, 4]`
- Tuples work the same as lists except they use parentheses
- Remember that lists are mutable (can be altered by changing, adding, or removing objects)
- Tuples are not mutable
- Lists and tuples hold (or contain) objects so are sometimes referred to as containers
- They can also be called collections since they hold a collection of objects
- Another useful data type that is also a container or collection is the *dictionary*

Quick review of list creation

```
>>> num_list = [1, 5, 42, 2.78, 99]
>>> string_list = ['Hello', 'Goodbye', 'MECH', 'Ferris', 'Bulldog']
```

2 An Introduction to and Creating Dictionaries

Dictionaries (like lists and tuples) can be used to contain or collect a variety of objects. The primary difference between dictionaries and lists is how objects are stored and accessed. Dictionaries do not rely upon position to access objects, they use *keys*.

- Dictionaries are mutable
- Items in dictionaries are stored using key-value pairs and the values are accessed via the keys

- Dictionary keys...
  - Must be unique within each dictionary because values are linked to the keys
  - Must be immutable objects such as...
    - Strings
    - Integers
    - Floats
    - Tuples
- Values can be nearly any object type, including...
  - Integers
  - Floats
  - Strings
  - Lists
  - Tuples
  - Other dictionaries

2.1 Creating Dictionaries

Dictionaries are enclosed with curly braces `{}` instead of square brackets like lists. Each of the following methods for creating dictionaries is demonstrated in code cells.

- An empty dictionary can be created one two ways...
  - Assign a set of empty curly braces to a variable name, i.e. `MECH_classes = {}`
  - Using the `dict()` function, i.e. `my_dict = dict()`
- Create a dictionary by enclosing `key:value` pairs inside of curly braces (see code block)
- Create a dictionary with key-value pairs as a list of lists (or tuples) in the `dict()` function
- Add key-value pairs to a dictionary by indexing the dictionary name with a key and assigning it a value
- Zip lists of keys and values and place into the `dict()` function
- Assign values to "simple" keys inside of a `dict()` function

Creating an empty dictionary and adding items via indexing

```
MECH_classes = {}
MECH_classes['MECH 111'] = 'MET Seminar'
MECH_classes['MECH 122'] = 'Computer Apps 1'
print(MECH_classes)
```

```
{'MECH 111': 'MET Seminar', 'MECH 122': 'Computer Apps 1'}
```

Creating a dictionary using curly braces and `key:value` pairs

```
MECH_classes = {'MECH 111':'MET Seminar',
                'MECH 122':'Computer Apps 1'}
print(MECH_classes)
```

```
{'MECH 111': 'MET Seminar', 'MECH 122': 'Computer Apps 1'}
```

Creating a dictionary using `dict()` and lists of key, value pairs

```
MECH_classes = dict([['MECH 111', 'MET Seminar'],
                     ['MECH 122', 'Computer Apps 1']])
print(MECH_classes)
```

```
{'MECH 111': 'MET Seminar', 'MECH 122': 'Computer Apps 1'}
```

Creating a dictionary by zipping a list of keys and a list of values

```
course_numbers = ['MECH 111', 'MECH 122']
course_names = ['MET Seminar', 'Computer Apps 1']
MECH_classes = dict(zip(course_numbers, course_names))
print(MECH_classes)
```

```
{'MECH 111': 'MET Seminar', 'MECH 122': 'Computer Apps 1'}
```

Simple keys are strings that have no spaces. They can be treated like variable names during dictionary creation.

Creating a dictionary with "simple" keys

```
MECH_faculty = dict(Brady='JOH 422',
                    Hollenbeck='JOH 412',
                    Stein='JOH 407',
                    Wiltshire='JOH 411')
print(MECH_faculty)
```

```
{'Brady': 'JOH 422', 'Hollenbeck': 'JOH 412', 'Stein': 'JOH 407',
 ↪  'Wiltshire': 'JOH 411'}
```

Not all dictionaries need to use strings as keys. The following dictionary definition is valid even though the keys are not all the same object type. The important thing is that all of the keys are immutable objects.

A dictionary with different types of keys

```
import math
my_dict = {
    1: 'one',
    2: 'two',
    math.e: 'e',
    3: 'three',
    'pi': math.pi,
    4: 'four',
    5: 'five',
    6: 'scared',
    7: 'ate nine',
    8: 'eight',
    9: None
}
print(my_dict)
```

```
{1: 'one', 2: 'two', 2.718281828459045: 'e', 3: 'three', 'pi':
↪ 3.141592653589793, 4: 'four', 5: 'five', 6: 'scared', 7: 'ate nine',
↪ 8: 'eight', 9: None}
```

2.2 Why Use Dictionaries?

Question: "Can't we just use lists instead of dictionaries?" *Answer*: "Yes, most of the time". Lists are used in many places where dictionaries can be used, and vice versa. Dictionaries are often easier, more efficient, or more effective to use. See the following examples...

- Example 1 – accessing material properties of common engineering materials
  - Would likely require lots of indexing and searching if lists were used
  - Material names can be used as keys to return all of the property values
  - Nested dictionaries could be implemented such that all properties for a material could be accessed by their names, i.e. `materials['steel']['poissons ratio']`

The following small material property dictionary uses lists for the material properties (just the USCS and SI yield strengths in ksi and MPa at this time). One would need to have knowledge of which list index is associated with each property to use it.

Simple material property dictionary

```
material = {"Steel ASTM-A36":[36, 250],
            "Aluminum 2014-T6":[60, 410],
            "Bronze cold-rolled":[75, 772],
            "Nickel Alloy":[60, 414]}
```

- Example 2 – turn a user input string into Morse Code
  - Create a dictionary with keys that are letters of the alphabet, numbers, and other symbols
  - Values are strings of dots and dashes
  - Loop through a string and access the dictionary by current character to translate to Morse code

The code block below defines a function named `string_to_morse(string)` that prints strings as Morse Code.

Morse Code converter featuring a dictionary

```
def string_to_morse(string):
    # create the dictionary
    to_morse = {'A':'.-', 'B':'-...',
                'C':'-.-.', 'D':'-..', 'E':'.',
                'F':'..-.', 'G':'--.', 'H':'....',
                'I':'..', 'J':'.---', 'K':'-.-',
                'L':'.-..', 'M':'--', 'N':'-.',
                'O':'---', 'P':'.--.', 'Q':'--.-',
                'R':'.-.', 'S':'...', 'T':'-',
                'U':'..-', 'V':'...-', 'W':'.--',
                'X':'-..-', 'Y':'-.--', 'Z':'--..',
                '1':'.----', '2':'..---', '3':'...--',
                '4':'....-', '5':'.....', '6':'-....',
                '7':'--...', '8':'---..', '9':'----.',
                '0':'-----', ',':'--..--', '.':'.-.-.-',
                '?':'..--..', '/':'-..-.', '-':'-....-',
                '(':'-.--.', ')':'-.--.-'}
    test_string = string.upper()
    morse_string = ""
    for character in test_string:
        morse_string += to_morse.get(character, " ") + " "
    return morse_string
```

```
>>> print(string_to_morse("Python is awesome"))
.--. -.-- - .... --- -.   .. ...   .- .-- . ... --- -- .
```

3 Accessing Dictionary Values

Dictionary values are accessed using their keys, not by numeric indices like lists and tuples.

- Use `MECH_faculty['Brady']` to access Mr. Brady's office location from a previous dictionary
- Trying to use a numeric index to access a value causes an error
- Prior to *Python* 3.7 the objects collected in dictionaries were stored with no particular order

- As of 3.7 objects are stored in the created order, but numeric indexing is still not allowed
- Accessing *Python* dictionary values is generally more efficient than searching and indexing lists
- Multiple sets of square brackets with keys or indexes are used to access values that are inside lists, tuples, or other dictionaries within a dictionary
- Example: `material["Steel ASTM-A36"][1]` will return the second property (SI yield strength) for ASTM-A36 steel from the earlier dictionary
- Trying to access a key that does not exist in a dictionary will result in an a `KeyError`
  - Test to see if a key is present in a dictionary using the `in` operator before accessing it
  - There is a way to attempt to retrieve a value for a key that does not exist without generating an error (later)
- Determine how many key-value pairs exist in a dictionary using the `len()` function

Accessing office locations from the `MECH_faculty` dictionary

```
>>> MECH_faculty['Stein']
'JOH 407'
>>> MECH_faculty['Wiltshire']
'JOH 411'
>>> MECH_faculty['Hollenbeck']
'JOH 412'
>>> MECH_faculty['Brady']
'JOH 422'
>>> MECH_faculty[0]  # will throw an error since 0 is not a key
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
```

Accessing material properties from the `material` dictionary

```
>>> material['Nickel Alloy']
[60, 414]
>>> material['Aluminum 2014-T6'][0]  # USCS yield strength
60
```

Accessing classes from the `MECH_classes` dictionary

```
>>> MECH_classes['MECH 111']
'MET Seminar'
>>> MECH_classes['MECH 499']   # will throw an error since no MECH 499
↪  yet
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'MECH 499'
```

Check if a key is in a dictionary

```
>>> 'Stainless Steel' in material
False
>>> 'Bronze cold-rolled' in material
True
```

Number of keys in a dictionary

```
>>> len(material)
4
>>> len(MECH_classes)
2
>>> len(MECH_faculty)
4
```

4 Adding or Replacing Dictionary Values

Adding and replacing key-value pairs uses one of the same methods as was used for initially creating a dictionary. Add/replace by simply assigning a value to a dictionary key using square brackets, such as `my_dict['age'] = 55`.

- `MECH_classes['MECH 211'] = 'Fluid Mechanics'` adds the key:value pair `'MECH 211':'Fluid Mechanics'` to the `'MECH_classes'` dictionary
- If a key already exists, the existing value will be overwritten with the new value since there can only be one instance of a key in a dictionary

Adding a key-value pair and changing a value in `MECH_classes`

```
>>> print(MECH_classes)
{'MECH 111': 'MET Seminar', 'MECH 122': 'Computer Apps 1'}

>>> MECH_classes['MECH 211'] = 'Fluid Mechanics'
>>> print(MECH_classes)
{'MECH 111': 'MET Seminar', 'MECH 122': 'Computer Apps 1', 'MECH 211':
 ↪  'Fluid Mechanics'}

>>> MECH_classes['MECH 122'] = 'Computer Apps 1 for Technology'
>>> print(MECH_classes)
{'MECH 111': 'MET Seminar', 'MECH 122': 'Computer Apps 1 for Technology',
 ↪  'MECH 211': 'Fluid Mechanics'}
```

5 Nested Dictionaries

Nesting dictionaries within the material properties dictionary would make selecting a property more explicit. A user could not only access a material by its name, they could access a specific property and units type. The upcoming code block includes the following modifications to the `material` dictionary to demonstrate nesting.

- Assign dictionaries to each material with the keys `'yield strength'` and `'ultimate strength'`
- Assign a dictionary as a value to each of the strength keys with the keys `'uscs'` and `'si'`
- Assign values to each of the unit keys
- `material_2["Steel ASTM-A36"]["yield strength"]["si"]` will return the SI yield strength for ASTM-A36 steel

A better material dictionary with nesting

```
material = {
            "Steel ASTM-A36":{
                 "yield strength":{"uscs":36, "si":250},
                 "ultimate strength":{"uscs":60, "si":400}
                 },
            "Aluminum 2104-T6":{
                "yield strength":{"uscs":60, "si":410},
                 "ultimate strength":{"uscs":70, "si":480}
                 },
            "Bronze cold-rolled":{
                 "yield strength":{"uscs":75, "si":772},
                 "ultimate strength":{"uscs":100, "si":515}
                 },
            "Nickel Alloy":{
                 "yield strength":{"uscs":60, "si":414},
                 "ultimate strength":{"uscs":80, "si":552}
                 }
            }

matl_1 = material['Nickel Alloy']['yield strength']['si']
matl_2 = material['Steel ASTM-A36']['ultimate strength']['uscs']

print(f"The SI yield strength of Nickel Alloy is {matl_1} MPa")
print(f"The USCS ultimate strength of ASTM-A36 Steel is {matl_2} ksi")
```

```
The SI yield strength of Nickel Alloy is 414 MPa
The USCS ultimate strength of ASTM-A36 Steel is 60 ksi
```

6 Dictionary Methods

Most dictionary methods will act "in-place", modifying the existing dictionary, instead of creating copies of the dictionary. The code block below prints the directory of the dictionary methods.

Directory of dictionary methods

```
>>> print([n for n in list(dir(dict)) if "__" not in n])
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',
↪  'setdefault', 'update', 'values']
```

- `dict.clear()` removes all key-value pairs; leaving an empty dictionary
- `my_dict.copy()` creates an unlinked copy of the dictionary `my_dict`
- `my_dict.keys()` returns all of the keys; convert the returned keys to a list using `list()`
- `my_dict.values()` returns all of the values; convert the returned values to a list using `list()`
- `my_dict.items()` returns all of the key-value pairs as tuples; use `list()` to convert to a list of tuples where each tuple contains the key-value pairs

Keys, values, and items in the `MECH_classes` dictionary

```
>>> list(MECH_classes.keys())
['MECH 111', 'MECH 122', 'MECH 211']

>>> list(MECH_classes.values())
['MET Seminar', 'Computer Apps 1 for Technology', 'Fluid Mechanics']

>>> list(MECH_classes.items())
[('MECH 111', 'MET Seminar'), ('MECH 122', 'Computer Apps 1 for
↪  Technology'), ('MECH 211', 'Fluid Mechanics')]
```

- `my_dict.get("a")` attempts to get the value associated with the key `"a"` from the dictionary
  - Returns the value if `"a"` is present
  - Returns `None` if the key is not present
  - Add a second argument that changes the returned value if the requested key is not present, for example, `my_dict.get("a", "Not present")`
  - This method prevents an error message when requesting a value from a dictionary for a key that is not present

Using `.get()` to retrieve a value for a key that exists

```
>>> # get 'Brady' from 'MECH_faculty' and return None if not
>>> MECH_faculty.get('Brady')
'JOH 422'
```

Using `.get()` to retrieve the values for keys that do not exist

```
>>> # get 'Drake' from 'MECH_faculty' and return 'Retired' if not
>>> MECH_faculty.get('Drake', 'Retired')
'Retired'

>>> # get 'Moore' and return 'Not faculty' if not
>>> MECH_faculty.get('Moore', 'Not faculty')
'Not faculty'
```

- `my_dict.pop("a")` returns the value associated with the key `"a"` and removes `"a"` and its value from the dictionary
- `my_dict.popitem()` returns the value associated with the last key and removes that key and value from the dictionary
  - Last key is removed when using *Python* 3.7 and later
  - Earlier versions of *Python* selected a random key from the dictionary
- `my_dict.setdefault('key')` and `my_dict.setdefault('key', 'value')` both try to add `'key'` to the dictionary
  - If key `'key'` already exists, its value is returned
  - If `'key'` doesn't already exist...
    - `None` is assigned to `'key'` when only one argument is provided
    - `'value'` is assigned to `'key'` when the second argument is included

Using `.setdefault()` to retrieve a value or create a new key-value pair

```
>>> print(MECH_classes)
{'MECH 111': 'MET Seminar', 'MECH 122': 'Computer Apps 1 for Technology',
 ↪  'MECH 211': 'Fluid Mechanics'}

>>> MECH_classes.setdefault('MECH 211')   # this key exists already
'Fluid Mechanics'

>>> MECH_classes.setdefault('MECH 490')   # this key does not exist yet
>>> print(MECH_classes)
{'MECH 111': 'MET Seminar', 'MECH 122': 'Computer Apps 1 for Technology',
 ↪  'MECH 211': 'Fluid Mechanics', 'MECH 490': None}

>>> MECH_classes.setdefault('MECH 498', 'Senior Project 1')
'Senior Project 1'

>>> print(MECH_classes)
{'MECH 111': 'MET Seminar', 'MECH 122': 'Computer Apps 1 for Technology',
 ↪  'MECH 211': 'Fluid Mechanics', 'MECH 490': None, 'MECH 498': 'Senior
 ↪  Project 1'}
```

- `new_dict = dict.fromkeys(sequence, value)` creates a dictionary named `new_dict`
  - The second argument, `value`, is optional
  - Every item in `sequence` (a list or tuple) will be used as keys
  - All keys will be assigned `value` (when provided) or `None`
- `my_dict.update(other_dict)` updates the values in `my_dict` using the key-value pairs from `other_dict`
  - Key-value pairs from `other_dict` that do not exist in `my_dict` will be added to `my_dict`
  - If a key from `other_dict` already exists in `my_dict`, its value in `my_dict` will be replaced by the one from `other_dict`

Using `.update()` together with `dict.fromkeys()` to add more classes

```
>>> # add MECH 212, MECH 222, and MECH 223 with values of None
>>> MECH_classes.update(dict.fromkeys(['MECH 212',
...                                    'MECH 222',
...                                    'MECH 223']))
>>> print(MECH_classes)
{'MECH 111': 'MET Seminar', 'MECH 122': 'Computer Apps 1 for Technology',
 ↪  'MECH 211': 'Fluid Mechanics', 'MECH 490': None, 'MECH 498': 'Senior
 ↪  Project 1', 'MECH 212': None, 'MECH 222': None, 'MECH 223': None}
```

A common use for a dictionary is to count the number of occurrences of each character in a string or numbers in a list. The following function definition will count the number of each unique character in the string and assign the counts to a dictionary.

Counting the number of each character in a string

```
def count_chars(string):
    string = string.lower()
    count_dict = {}
    for char in string:
        if char in count_dict:
            count_dict[char] += 1
        else:
            count_dict[char] = 1
    return count_dict

print(count_chars("I'm a lumberjack and I'm okay!"))
```

```
{'i': 2, "'": 2, 'm': 3, ' ': 5, 'a': 4, 'l': 1, 'u': 1, 'b': 1, 'e': 1,
 ↪  'r': 1, 'j': 1, 'c': 1, 'k': 2, 'n': 1, 'd': 1, 'o': 1, 'y': 1, '!':
 ↪  1}
```

A better version of the `count_chars(string)` function would be to use `count_dict.get(char, 0)`

instead of the `if-else` statement block. Using a default value of `0` in the `.get()` method causes it to add a key for `char` and assign it the value 0 if `char` is not already a key. The following code block demonstrates the changes to the function.

Improved function to count each character in a string using `.get()`

```
def count_chars(string):
    string = string.lower()
    count_dict = {}
    for char in string:
        count_dict[char] =  count_dict.get(char, 0) + 1
    return count_dict

print(count_chars('The quick brown fox jumps over the lazy dog'))
```

```
{'t': 2, 'h': 2, 'e': 3, ' ': 8, 'q': 1, 'u': 2, 'i': 1, 'c': 1, 'k': 1,
↪  'b': 1, 'r': 2, 'o': 4, 'w': 1, 'n': 1, 'f': 1, 'x': 1, 'j': 1, 'm':
↪  1, 'p': 1, 's': 1, 'v': 1, 'l': 1, 'a': 1, 'z': 1, 'y': 1, 'd': 1,
↪  'g': 1}
```

The string used above has at least one of each letter from the English alphabet. If you assign the function call to a variable, like `string_count`, then use `sorted(string_count)` it is fairly easy to verify this. Sorting a dictionary sorts by and only shows the keys.

Sorting the keys in a dictionary using `sorted()`

```
string_count = count_chars('The quick brown fox jumps over the lazy dog')
print(sorted(string_count))
```

```
[' ', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
↪  'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

7 Sets

Sets are unordered collections of unique values that are related to, and sometimes mistaken for, dictionaries. They do use curly braces, like dictionaries, to enclose the values.

- Sets do not contain key-value pairs like dictionaries
- Sets can be created from any of the following by using the `set()` function
  - Strings
  - Lists
  - Tuples
  - Dictionary keys
  - Dictionary values

- Nothing will be changed if a value is added to a set that already contains that value
- One good use for a set is to determine the unique characters in a string

Creating (and sorting) a set from a string

```
>>> string_set = set("the quick brown fox jumps over the lazy dog")
>>> print(string_set)
{'t', 'x', 'q', 'h', 'z', 'd', 'f', 'i', 'j', 'n', 'o', 'c', 'l', 'r',
 ↪  'p', 'u', 'y', 'a', 'g', 'v', 'w', 'm', 's', ' ', 'b', 'k', 'e'}

>>> print(sorted(string_set))
[' ', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
 ↪  'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

>>> len([letter for letter in string_set if letter.isalpha()])
26
```

8 Reading and Writing from/to Files

Items in lists, tuples, and dictionaries only remain available until a script finishes executing. Sometimes the desire is to save values for later or use previously saved information, such as...

- Test results saved from an instrument or machine
- Tables of material properties
- Steam tables
- A dictionary of books, articles, or standards
- Any text file

Python has built-in functions and methods for reading and writing information from/to files. Examples here use text files that can be opened with any standard text editor.

8.1 Opening a File for Reading or Writing

It is assumed that all files are located in the same directory as the script or function reading from or writing to the files. One method is to open a file is using the `open()` function and assigning it to a variable.

- `my_file = open("data file.txt")` will open the file `"data file.txt"` for reading and assign it to `my_file`
- A mode argument can be added to the function call to explicitly state how the file will be used
  - Use `"r"` for reading (this is the default)
  - Use `"w"` for writing
  - Use `"a"` for appending
  - Use `"r+"` for reading and writing
- After a file is opened for reading or writing, it needs to be closed it, i.e. `my_file.close()`

A better, more robust method is to use the `with` statement in conjunction with the `open()` function. The `with` statement is referred to as a *context manager* in *Python*.

- `with open("data file.txt", "r") as my_file:`
- It does exactly the same thing as the previous method
- Any commands done with the file need to be indented under the `with` line
- The file is automatically closed as soon as the code block indented under the `with` block is finished
- Using `with` to open a file means that the `.close()` method does not need to be used

8.2 Reading a File's Contents

There are two methods for reading the contents of an opened file that should meet most needs; `file.read()` and `file.readlines()`. Keep in mind that the original file is not altered in any way when it is read.

- `file.read()` will read an entire text file as a single long string
  - Every location that the file had a new line, the string will have the newline character `\n` added
  - Assign the `file.read()` to a variable to work with the string
- `file.readlines()` is similar except that each line of the text file becomes an item in a list
- It is best to perform any operations beyond reading outside of a `with` context (if used)

Reading a text file using `.read()`

```
# the `./` with the file name means to check the current directory
with open("./data file.txt") as my_file:
    file_string = my_file.read()

print(file_string)
```

```
Hello, World!
Python is awesome!
```

Reading a text file using `.readlines()` this time

```
with open("./data file.txt") as my_file:
    file_list = my_file.readlines()

print(file_list)
```

```
['Hello, World!\n', 'Python is awesome!']
```

8.3 Writing to a File

Sometimes the desire is to perform tasks/calculations and save the results to a file instead of printing them. Actions for writing files are similar to reading a file; open a file and write to it.

- Must explicitly set the mode to either `"w"` or `"a"`
  - If `"w"` or `"a"` are used and the file does not already exist, it will be created
  - If `"w"` is used and the file already exists, its content will be deleted without warning
  - If `"a"` and the file exists, the existing file will be appended
- Once a file is open either the `file.write()` or `file.writelines()` methods can be used
- It is important to note that only strings can be written; *Python* will not write integers or floats (or anything else) unless converted to strings first
- The `file.write()` method requires a string as an argument and writes the string to the open file
- The code block below opens `"new file.txt"` and writes two lines to it
- The `\n` at the end of each print statement is necessary for each string to be on separate lines

Writing to a file with `file.write()`

```
with open("new text file.txt", "w") as new_file:
    a = new_file.write("Hey, new file.\n")
    b = new_file.write(f"{21 * 2}")
```

- The `file.writelines()` method requires a list of strings as an argument
- All of the strings in the list will be written one after the other to the file
- If a string does not end in `\n`, then the next string will start on the same line
- The following code shows how a list of strings can be used to create a new text file

Writing to a file with `file.writelines()`

```
string_list = ["My name is Brian.\n", "I work at Ferris."]
with open("brian file.txt", "w") as new_file:
    new_file.writelines(string_list)
```

Reading the two files that were written

```
with open("./new text file.txt") as file1:
    print(file1.read())
with open("./brian file.txt") as file2:
    print(file2.read())
```

```
Hey, new file.
42
My name is Brian.
I work at Ferris.
```

8.4 Reading CSV Files

CSV (comma separated values) files are special forms of text files; they essentially just have values separated by commas on each line. All modern spreadsheet applications can be used to create/export CSV files.

Contents of `my_csv_file.csv`

```
"x","y"
0,0
1,2
3,6
4,8
5,10
```

The standard `open()` function with `file.read()` or `file.readlines()` plus some other code can be used to read CSV files. The following code block opens and reads the lines of a CSV file. It then strips (removes) newline characters from the ends of each line and splits each line at the the commas. Numeric values are converted to floats.

Reading and processing values from `my_csv_file.csv`

```
with open("my_csv_file.csv", "r") as csv_file:
    csv_list = csv_file.readlines()

x = []
y = []
for string in csv_list[1:]:
    new_string = string.strip("\n")
    xy_pair = [float(n) for n in new_string.split(",")]
    x.append(xy_pair[0])
    y.append(xy_pair[1])

headings = csv_list[0].strip("\n").strip('"').split('","')

print(headings)
print(x)
print(y)
```

```
['x', 'y']
[0.0, 1.0, 3.0, 4.0, 5.0]
[0.0, 2.0, 6.0, 8.0, 10.0]
```

The csv module was designed to handle this specific task more gracefully.

- Use open() and the with statement to open a CSV file and assign it a name
- Use csv.reader() with the assigned name as an argument and assign the result to a name
  - Add the keyword argument quoting=csv.QUOTE_NONNUMERIC
  - The quoting keyword tells the reader how to handle quoted items
  - In this case, it converts non-quoted items into floats
- The new CSV object can be converted to a list so the required values can be extracted
- In the code below two list comprehensions are used to extract the x and y values and place them in their own lists

Using the csv module

```
import csv
with open("my_csv_file.csv") as csvfile:
    csv_reader = csv.reader(csvfile, quoting=csv.QUOTE_NONNUMERIC)
    csv_list = list(csv_reader)
headings = csv_list[0]
x = [n[0] for n in csv_list[1:]]
y = [n[1] for n in csv_list[1:]]

print(headings)
print(x)
print(y)
```

```
['x', 'y']
[0.0, 1.0, 3.0, 4.0, 5.0]
[0.0, 2.0, 6.0, 8.0, 10.0]
```

8.5 Using *Pandas* to Read a CSV File

People who use *Python* to work with large data sets typically use the *Pandas* library https://pandas.pydata.org/. This library of modules makes working with CSVs of any size very easy. The primary data structure of *Pandas* is the *DataFrame*.

- The following code block will import *Pandas* and use .read_csv() to create a DataFrame named csv_df from "my_csv_data.csv"
- Once csv_df is created, its headings can be converted to a list using list(csv_df)
- DataFrames use the heading names from the CSV as indexes for the DataFrame (like dictionaries)
- In this case the headings are "x" and "y"
- csv_df["x"] will return all of the values from the "x" column as a *Pandas* series
- Use the list() function to convert the series to a list, i.e. x = list(csv_df["x"])

Reading a CSV file using *Pandas*

```
import pandas as pd
csv_df = pd.read_csv("my_csv_file.csv")
headings = list(csv_df)
x = list(csv_df['x'])
y = list(csv_df['y'])

print(headings)
print(x)
print(y)
```

```
['x', 'y']
[0, 1, 3, 4, 5]
[0, 2, 6, 8, 10]
```

The *Pandas* approach definitely took fewer lines of easier to follow code, but required more computing overhead due to importing a fairly sizable library. However, with that overhead comes a lot of power to work with data sets both large and small.

9 Some Creative Commons Reference Sources for This Material

- *Think Python 2nd Edition*, Allen Downey, chapters 11 and 14
- *The Coder's Apprentice*, Pieter Spronck, chapter 13, 16, and 26
- *A Practical Introduction to Python Programming*, Brian Heinold, chapters 11 and 12
- *Algorithmic Problem Solving with Python*, John Schneider, Shira Broschat, and Jess Dahmen, chapters 10 and 14